NG2C: N-Generational Garbage Collector for Big Data Memory Management

Rodrigo Bruno, Paulo Ferreira

{rodrigo.bruno,paulo.ferreira}@inesc-id.pt

INESC-ID - Instituto Superior Técnico, ULisboa

Motivation

- Applications use large amounts of memory to enable fast data access
  - Eg: memory caches used in Cassandra or RDDs on Spark

Motivation

- Applications use large amounts of memory to enable fast data access
  - Eg: memory caches used in Cassandra or RDDs on Spark

- Having lots of data in memory puts too much stress on current memory management technologies, in particular, the Garbage Collector (GC);

Motivation

- Applications use large amounts of memory to enable fast data access
  - Eg: memory caches used in Cassandra or RDDs on Spark

- Having lots of data in memory puts too much stress on current memory management technologies, in particular, the Garbage Collector (GC);

  - **! Leads to big application pauses which compromise performance and responsiveness !**

Goals

1. Minimizing GC stop-the-world pauses. **How?**

Goals

1. Minimizing GC stop-the-world pauses. **How?**
2. Avoiding object copy within the heap. **Why?**

Goals

1. Minimizing GC stop-the-world pauses. **How?**

2. Avoiding object copy within the heap. **Why?**

    - We found that the cost of GC stop-the-world pauses is mostly dominated by the number (and size) of objects to copy:

Goals

1. Minimizing GC stop-the-world pauses. **How?**

2. Avoiding object copy within the heap. **Why?**

    - We found that the cost of GC stop-the-world pauses is mostly dominated by the number (and size) of objects to copy:

        - Promotion (moving objects from young to old generation)

        - Compaction (compacting objects within the old generation)

Solution

- Avoid copying objects in memory. **How?**

Solution

- Avoid copying objects in memory. **How?**
- Replacing the current heap layout of two generations (young and old) by an arbitrary number of generations. **How?**

Solution

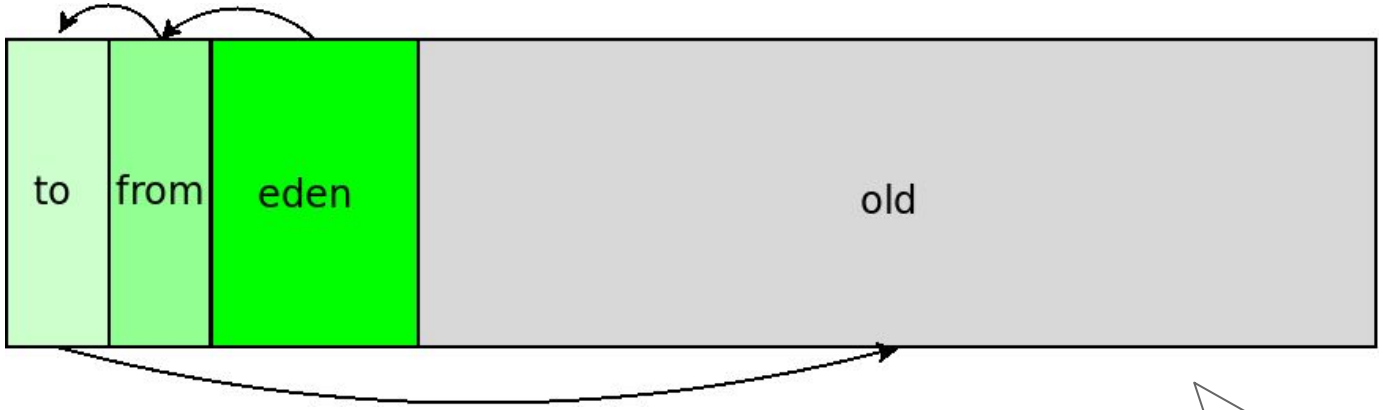
- Avoid copying objects in memory. **How?**
- Replacing the current heap layout of two generations (young and old) by an arbitrary number of generations. **How?**
  - Giving the programmer the power to:
    - Create and collect specific generations (run time);
    - To allocate objects directly in a specific generation;

Solution

- Avoid copying objects in memory. **How?**
- Replacing the current heap layout of two generations (young and old) by an arbitrary number of generations. **How?**
  - Giving the programmer the power to:
    - Create and collect specific generations (run time);
    - To allocate objects directly in a specific generation;
- Each generation should contain objects with similar expected life-cycles
  - All objects in a generation are expected to die about the same time
  - Eg: all objects stored in a cache die when the cache is flushed;
  - Eg: all objects created to handle a specific computation task die when the task is finished.

Solution

- x Objects allocated in the Eden are copied several times;

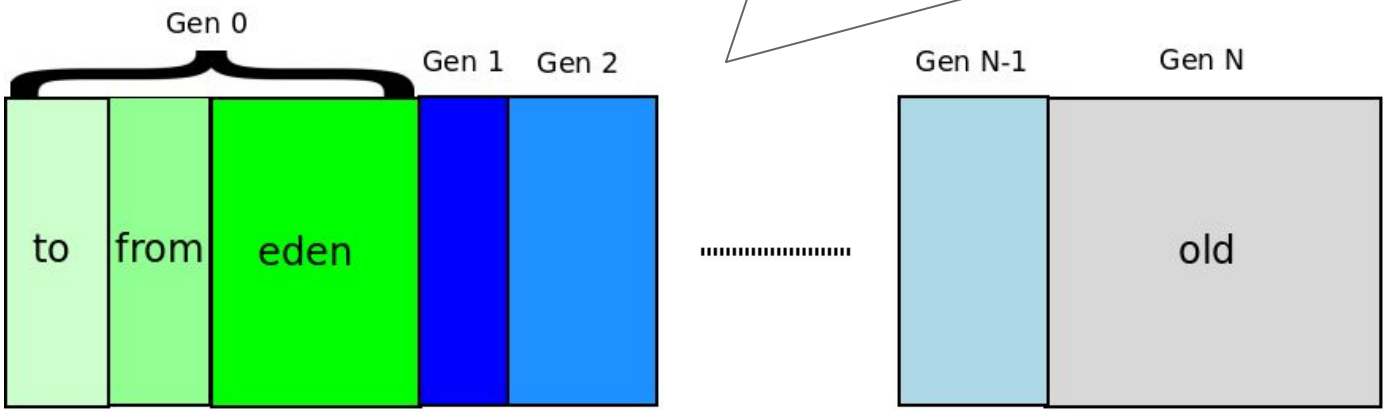


2-Generation (traditional) Heap Layout

- x Old generation is hard to collect, potentially leading to full GCs;

Solution

✓ Objects allocated in specific generation, near other objects with the same life cycle.



N-Generation Heap Layout

✓ Heap is organized according to object life cycle. Easy to collect.

Code Sample

```
public void runTask() {
    System.newGen();
    while (running) {
        DataChunk data = new @Gen DataChunk();
        initializeData(data);
        doComplexProcessing(data);
    }
    System.collectGen();
}
```

Creates a new generation. Allocations with the @Gen annotation will go directly to this generation.

Special annotation for allocating object in specific generation (other than Eden).

Creates a new epoch in the current generation. Memory previously allocated s now ready to be collected.

Solution

- Is it a good idea to ask the programmer to give hints to the GC?

Solution

- Is it a good idea to ask the programmer to give hints to the GC?
  - For most applications: NO!

Solution

- Is it a good idea to ask the programmer to give hints to the GC?
  - For most applications: NO!
  - For applications with strict performance requirements: YES!

Solution

- Is it a good idea to ask the programmer to give hints to the GC?
  - For most applications: NO!
  - For applications with strict performance requirements: YES!
    - Most applications already resort to several tricks to circumvent the GC (eg: using offheap memory, keep memory objects bounded, etc…)

Solution

- Is it a good idea to ask the programmer to give hints to the GC?
  - For most applications: NO!
  - For applications with strict performance requirements: YES!
    - Most applications already resort to several tricks to circumvent the GC (eg: using offheap memory, keep memory objects bounded, etc…)
    - Places where generations are allocated and collected are usually well defined;

Preliminary Results

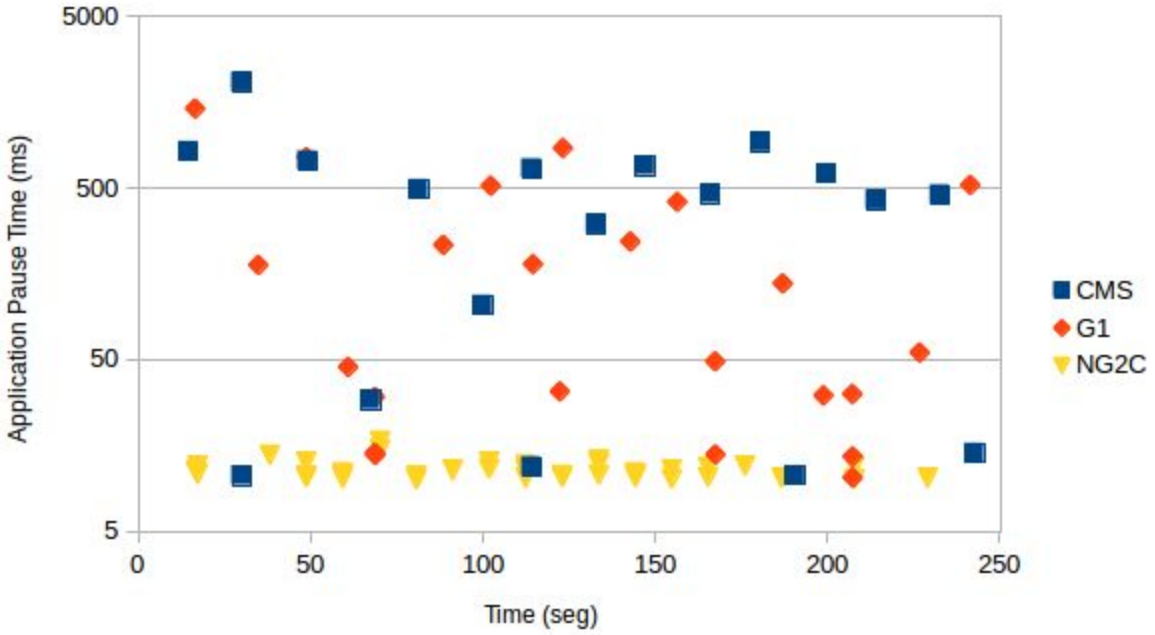
- Simple application (very similar to code sample):
  - 4 threads processing tasks;
  - Each task has 1 GB of data to process;
  - Total working set of 4 GBs.

- Three collectors used:
  - Concurrent Mark-and-Sweep (default GC for OpenJDK < 9)
  - Garbage First (default GC for OpenJDK >= 9)
  - N-Generational Garbage Collector (our collector)

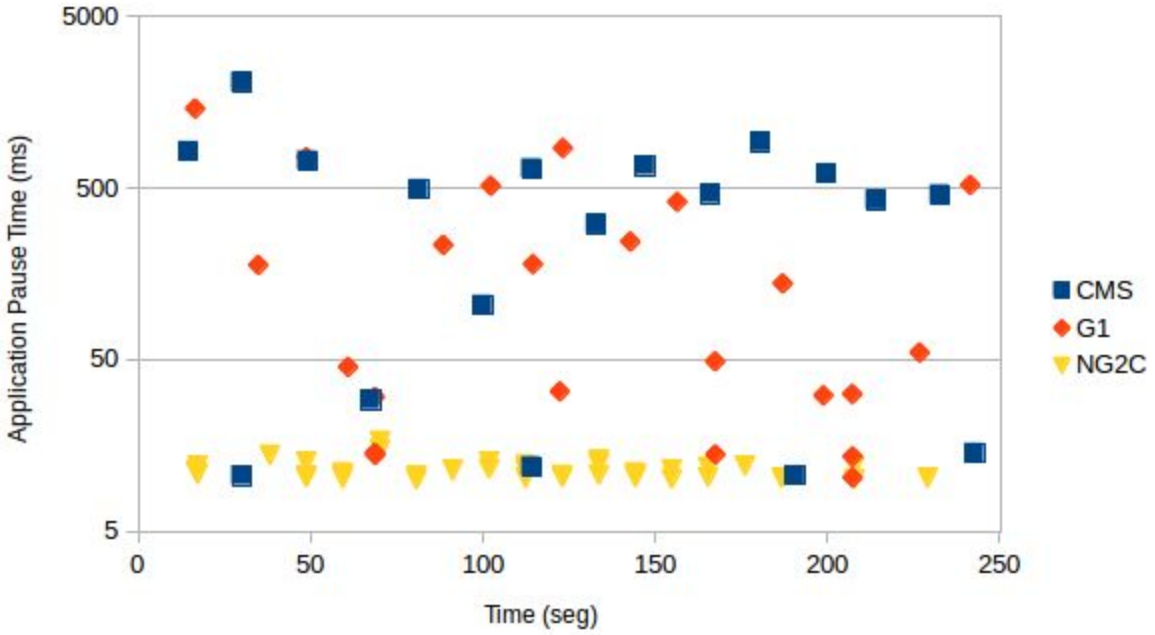
- Both CMS and G1 with young generation sizes of 8 GBs (twice the working set). NG2C with 1 GB.

- Heap size fixed at 12 GBs for all collectors.

Preliminary Results



Preliminary Results



Percentiles	50	75	90	99	99.9	99.99
CMS (ms)	461	670	853	1873	2048	2065
G1 (ms)	94	241	588	838	853	854
NG2C (ms)	11	12	14	16	16	16

Thank you for your time.
Questions?
Suggestions?

Rodrigo Bruno
email: rodrigo.bruno@tecnico.ulisboa.pt
webpage: www.gsd.inesc-id.pt/~rbruno